Overview of Secure Controls Framework Implementation

Version 2. June 22, 2004
A. D. Petrov, apetrov@fnal.gov

The Secure Controls Framework (SCF) was proposed in [1] as a way to improve the communication between Java™ controls applications and server-side data providers over RMI. This document gives an overview of the first productional version of SCF released on June 21, 2004.

Architecture

SCF consists of:

1. Client tier, which initiates and maintains connections and provides a high-level API for the user applications;
2. Data server, which accepts incoming connections, performs authentication of the clients, distributes and handles data requests;
3. Set of reusable graphical components that can be used to create client-side GUI;
4. Supplementary Kerberos module that provides user authentication based on Kerberos V5 protocol.

Client and server parts communicate through Remote Method Invocation protocol (RMI) over TCP/IP. On the transport level, Generic Security Service (GSS) API with Kerberos V5 mechanism is used to provide per-message integrity and (potentially) encryption.

Purpose

SCF provides two services:

1. Access to some common repositories: ACNET device database, parameter page database, and Universal Repository of Serialized Objects. On the client side, generic Java Naming & Directory API (JNDI) is used to get the content;
2. Bridge to the Controls Data Acquisition System (DAQ). A custom set of classes is used to define the data acquisition job and to hold the data.

SCF server has pluggable architecture and the number of services can be easily extended. The naming service is also pluggable and allows deploying of additional data providers.

Security

The major reason of SCF development was to resolve security problems existed in the current DAQ implementation in order to allow the clients to read and set data from many locations. SCF provides the following features:

1. Kerberos V5 is used for the user authentication on the client side;
2. JGSS API is used to establish a common security context between the client and the server;
3. Per-message security services are applied to the data traveling over the established security context in both directions in order to provide data integrity;
4. A decision whether the client is authorized to do some operation is made on the server. The client is responsible only for establishing of GSS context;
5. Each server-side service (exported remote interface) has its own fixed TCP/IP port;
6. The client-side code has no direct connection to the database or other critical resources;
7. All the SCF code that is taking care of security is based on a generic implementation from the publicly released SDK. Our extensions and changes are as minimal as possible;
8. All user logins are logged in the database.

Kerberos authentication is implemented in `tools.kerberos.base.Krb5`[*] class. Prior running any SCF client application, the user must obtain valid Kerberos ticket through some external utility, such as kinit or Leash32. The special procedure for exempt users located in the Main Control Room is described below. The Kerberos module can read the ticket from various types of caches.

The common security context is established in `scf.remote.ClientContextImpl` and `scf.server.ClientDelegate` classes on the client and server sides, correspondingly. As a result, both hosts obtain instances of `GSSContext` that are stored in `scf.remote.transport.SecurityController` on each side along with the remote port and internal context ID. `SecurityController` caches existing contexts and provide them for other parts of the system.

[*] Common package prefix `gov.fnal.controls` is omitted.

Per-message security is applied on the transport level. On both sides, RMI is using special extensions of output and input steams that can encrypt and decrypt the data with `GSSContext`'s `wrap` and `unwrap` methods. The output stream breaks the outgoing plaintext to the pieces of an appropriate size, gets `GSSContext` from the local `SecurityController`, `wrap`s it (converts to a cipertext), and creates a datagram that should be sent to the remote host. The datagram is the combination of ciphertext, its size, and used context ID. On the opposite side the input stream does the reverse: restores the datagram, gets local `GSSContext`, and `unwrap`s the data. The described mechanism is universal and can work without `GSSContext`, sending the plaintext directly.

The purpose of `SecurityController` is to cache locally all known `GSSContext`s and to associate them with the remote port, current thread (temporarily), and with a context ID. Generally speaking, a connection can have:

- No contexts at all — this takes place on the early stage during establishing of the context, and for exempt clients.
- One context —the regular way of communication.
- Two contexts — if the client was asked to re-authenticate, during establishing of new context.

The purpose of context ID is to give the input stream a clue which context to use for the particular datagram.

When the input stream successfully unwraps data, it temporarily associates the acting `GSSContext` with the current thread. If on the RMI level this message causes a method call, the executed code will be working in the same thread, so the method handler can check `SecurityController` in order to find out who is calling it. (The same mechanism is used in the generic implementation of `RemoteServer.getCurrentHost`).

Each server-side service uses remote user name and (may be) node name to properly authorize the clients. Naming providers store this data in the database. DAQ bridge has its own complicated security mechanism.

The described model does not work for the clients located in the Main Control Room that are not kerberized. To resolve this, a special group of *exempt* clients is described in the database, along with their IP addresses. All those hosts are supposed to be physically located in a trusted network, so their IPs can not be forged. If SCF server receives check-in request from the exempt host, it skips the explicit Kerberos authentication and defines the user name according to the host address.

The general security policy is defined in `scf.server.ZoneConfig` class. When a client is connected to the server, it is associated with a *zone*, according to its IP. Each zone has specific security policy attributes, such as allowing authentication without Kerberos login and default login lifetime.

Below is the connection scenario:

1. The client sends a check-in request to the server.
2. The server defines the client's IP zone. If logins without Kerberos are allowed in this zone and the client address is registered in the exempt host database, the server gets user name from there and notifies the client that it is registered under this name. Otherwise:
3. The server asks the client to provide its credentials.
4. The client gets cached Kerberos ticket, creates the initial GSS context, and sends corresponding token to the server.
5. The server accepts the token, creates and caches GSS context associated with the client, and sends another token to the client.
6. The client accepts the token, and caches GSS context created on step 4. Authentication is done.
7. The client starts sending service requests to the server.

Communication on steps 1–5 is not secure. Per-message security services are applied after GSS contexts are cached on both sides. If the client is authenticated without Kerberos, the communication is always insecure.

It is worth to say, that per-message security guarantees message integrity, but does not necessarily mean encryption. Basically, `GSSContext`'s `wrap` method adds a message integrity code (MIC) to the data, which is a sort of digital signature. The actual encryption is optional and can be requested by the client when it initiates the context.

Services

Naming service offers access to several repositories from the client side through JNDI:

- Device database (JNDI root = `device`) — ACNET device data: name, device index, properties, siblings, node name, and description. For each property: property index, array size, FTD, default event. Read only.
- Parameter page database (JNDI root = `page`) — tree of VMS parameter pages: page title, list of devices and text comments. Pages that are empty in the generic database are writable (so, the user can re-define its title and list of devices / comment). All others are read-only by default.
- Universal Repository of Serialized Objects (JNDI root = `object`) — allows to store random Java objects in serialized form. Has read and write access.

Objects retrieved from the device and parameter page databases can be directly used to create the data acquisition job.

Bridge to DAQ allows starting DAQ jobs on the server and receiving callbacks on the remote host. The definition of SCF job is the same as was used in DAQ (source / disposition / item / event), but all objects are re-defined and simplified to get rid of direct database access and to fit specific needs of client applications. Currently supported items are:

- Data sources:
  - `AcceleratorSource`
  - `ModelSource`
  - `UserSettingSource`
- Data disposition:
  - `AcceleratorDisposition`
  - `ModelDisposition`
  - `MonitorChangeDisposition`
- Events:
  - `DefaultDataEvent`
  - `OnceImmediateEvent`
  - `KnobSettingEvent`

In SCF, most of the listed items are just the placeholders mapped to corresponding DAQ objects on the server.

SCF equivalent of `DataItem` is more complicated: it can be either a single device with set of properties, or a composite device, which has a list of devices inside, including other composite devices. Device properties also act as containers that store values received from the data acquisition system.

Detailed description of DAQ bridge will be provided later in a separate user guide.

Secure Controls Framework also has a library of reusable graphical components for client application developers:

- Renderers that can format reading, setting, and status values in traditional way;
- SCF connection dialog integrated in Application Framework;
- `DeviceNameEditor`, `NamingCombo`, and `NamingTree` to browse the content of JNDI trees;
- `JobDataLabel` to show device readings and status;

- `JobDataSettingField` to do device settings;
- `JobDataTable` — a reusable implementation of traditional parameter page table.

Some components (`JobDataLabel`, `JobDataSettingField`, and `JobDataTable`) can be directly used as SCF job items.

Current Status

Currently, SCF is in productional beta release; detailed user guide is under development.

All parts of the projects can be found on Controls CVS or *\\daesrv\java* in three packages:

- `gov.fnal.controls.scf`
- `gov.fnal.controls.servers.scf`
- `gov.fnal.controls.tools.kerberos`

On the client side, one should use two jars in the classpath: `scf-client.jar` and `krb5.jar` (both can be found in *\\daesrv\java\jars\split*). No other libraries are required. In order to run the client, the user should get Kerberos ticket through kinit or Leash32 (both file cache and Windows memory cache are supported). Support of other Kerberos tools (especially for Mac) will be gradually provided.

SCF server is running on DPE08. At this time, the firewall does not pass SCF traffic; that is why this server is only accessible from the Controls network. The server status is available at *http://dpe08.fnal.gov:3333*.

So far, no serious or persistent errors were noticed on both server and client.

The reference implementation of SCF is Data Survey application, which is another version of Java Parameter Page (almost full equivalent). It can be found on *W* page of Application Index and started through Java Web Start.

Client code size estimate:

<u>In general:</u>

DAQ		SCF		Ratio
`govcore.jar`	4,971 K	`scf-client.jar`	223 K	
`jconn2.jar`	460 K	`krb5.jar`	45 K	
Total:	**5,431 K**	**Total:**	**268 K**	**5%**

Java Parameter Page *vs.* Data Survey:

Java Parameter Page		Data Survey		Ratio
`jpp.jar`	221 K	`survey.jar`	60 K	27%
`govcore.jar`	4,971 K	`scf-client.jar`	223 K	5%
`jconn2.jar`	460 K	`krb5.jar`	45 K	
`framework.jar`	365 K	`framework.jar`	365 K	100%
`af-bunch.jar`	826 K	`af-bunch.jar`	826 K	
Total:	**6,843 K**	**Total:**	**1,519 K**	**22%**

References

0. A. D. Petrov. Overview Of Secure Controls Framework Implementation. Version 1. Beams-doc-1040v1. *http://beamdocs.fnal.gov/cgi-bin/public/DocDB/ShowDocument?docid=1040*
1. A. D. Petrov. Proposals on a Secure RMI Connection for Client Applications. Beams-doc-953. *http://beamdocs.fnal.gov/cgi-bin/public/DocDB/ShowDocument?docid=953*
2. Java™ Authentication and Authorization Service (JAAS) Reference Guide. *http://java.sun.com/j2se/1.4.1/docs/guide/security/jaas/JAASRefGuide.html*
3. M. Upadhyay, R. Marty. Single Sign-on Using Kerberos in Java. Sun Microsystems, Inc. *http://java.sun.com/j2se/1.4.1/docs/guide/security/jgss/single-signon.html*
4. L. Gong, G. Ellison, M. Dageforde. Inside Java™ 2 Platform Security, Second Edition. Addison-Wesley ISBN 0-201-78791-1. p. 262.
5. Application Index web interface. *http://www-bd.fnal.gov/appix*
6. R. Lee. The JNDI Tutorial. *http://java.sun.com/products/jndi/tutorial*